2013

Robust Minkowski Sum Computation on the GPU

Elisha P. Sacks
Purdue University, eps@cs.purdue.edu

Min-HO Kyung
Ajou University, kyung@ajou.ac.kr

Victor Milenkovic
University of Miami, vjm@cs.miami.edu

Report Number:
13-001

Robust Minkowski Sum Computation on the GPU

Min-Ho Kyung
Department of Digital Media
Ajou University, Suwon, South Korea
kyung@ajou.ac.kr

Elisha Sacks
Computer Science Department
Purdue University
eps@cs.purdue.edu

Victor Milenkovic
Department of Computer Science
University of Miami
vjm@cs.miami.edu

March 13, 2013

Abstract

We present a robust convolution-based algorithm for Minkowski sums of polyhedra that is simpler, faster, and more memory efficient than our prior algorithm, and that has a provable error bound. We validate the algorithm on 45 inputs, including 9 highly degenerate ones. A GPU implementation exhibits an average speedup factor of 30.

1 Introduction

Minkowski sums of polyhedra are a core computational geometry concept with applications in solid modeling, packing, assembly, and robotics. Convolution-based algorithms [1] are efficient, but are hard to implement robustly because they have many special cases on degenerate input. Prior work provides a robust implementation of an inefficient algorithm, a robust partial implementation of a convolution algorithm, and approximate algorithms without error bounds or robustness guarantees (Sec. 2).

We [2] developed the first robust convolution-based algorithm (Sec. 3) and showed that it outperforms prior work. That algorithm has several limitations. The running time and the memory use are high on inputs with many degeneracies. The robustness technique requires custom logic for many types of degeneracy, can be inaccurate, and is ill-suited to GPU programming.

We present an improved algorithm that addresses these limitations (Sec. 4). The running time is reduced with improved geometric predicates. The memory use is reduced by removing intermediate geometry that cannot contribute to the output. We use a novel robustness strategy [3] to eliminate the custom logic and to enforce a user-specified error bound. We validate a CPU implementation of the algorithm on 45 inputs, including 9 highly degenerate ones (Sec. 6). A GPU implementation exhibits an average speedup factor of 30, excluding six inputs for which the intermediate geometry exceed its memory size. We conclude with plans for handling such inputs (Sec. 7).

2 Prior work

We discuss prior work on implementing Minkowski sums of polyhedra. We omit algorithms that are restricted to convex polyhedra because this case is well understood.

Hachenberger [4] uses exact evaluation to compute Minkowski sums. He decomposes the polyhedra into convex components, computes the component sums, and forms their union. A polyhedron with r reflex edges can have $\Omega(r^2)$ convex components. Computing the union of the component sums is a time and memory bottleneck. This cost plus the exact evaluation cost make the program prohibitively slow [5, 2].

Lien [6] computes Minkowski sums with a partial convolution algorithm combined with a collision detection algorithm. The algorithm is inefficient and is not robust.

Campen and Kobbelt [5] use a convolution algorithm to compute the outer boundary of the Minkowski sum. The algorithm has limited applicability because Minkowski sums with inner boundaries are common, e.g. when the inputs have inner cavities, in part layout, and in mechanical design. They use an efficient algorithm for exact predicate evaluation due to Shewchuk [7]. Since this algorithm does not support geometric constructions, they convert the input to a plane-based representation in which vertices are defined combinatorially. This representation also facilitates the handling of degenerate input. The accuracy of the input vertices is limited to five decimal digits by the requirements of Shewchuck's algorithm. Converting the Minkowski sum back to a five-digit vertex-based representation can cause it to self intersect.

Varadhan [8] computes approximate Minkowski sums using a volumetric grid [8]. A GPU implementation that computes the outer boundary of the Minkowski sum is fast [9]. The accuracy is limited by the volumetric resolution: the reported results have a resolution of 1024^3, which yields a 0.1% error. Increasing the resolution incurs a cubic running time penalty and is limited by the GPU memory size.

3 Minkowski sum algorithm

The Minkowski sum of point sets A and B is $A \oplus B = \{a + b | a \in A \wedge b \in B\}$. For polyhedra A and B with boundaries $\overline{A}$ and $\overline{B}$, $A \oplus B$ is a polyhedral region whose boundary is a subset of $\overline{A} \oplus \overline{B}$. The latter sum includes the Minkowski sums of every A vertex and B face, A face and B vertex, and A edge and B edge. We work with polyhedra with triangular faces; general faces must be triangulated. The sum of a vertex and a face is a triangle, while the sum of two edges is a parallelogram. Fig. 1 shows an example: the thick lines in the Minkowski sum boundary (b) delimit the portion that is generated by the sum polygons shown in the detail (c).

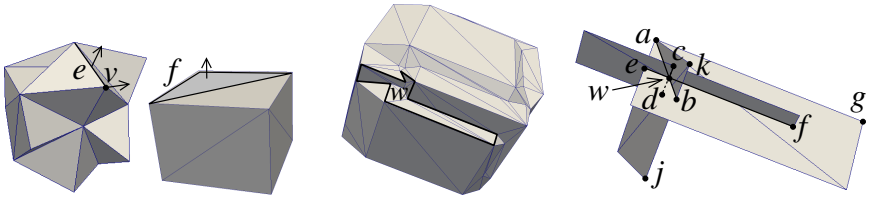


Figure 1: Polyhedra (a), Minkowski sum (b), and detail of sum polygons (c).

The vertices and edges of a sum polygon are called sum vertices (a and g in Fig. 1c) and sum edges (ag). The intersection point of a sum edge with a sum polygon is called an EP-vertex (b–f). The intersection segment of two sum polygons is called a PP-edge (ab, cd, and ef). Its endpoints are sum vertices or EP-vertices. The intersection point of three sum polygons (and of their three PP-edges) is called a PPP-vertex (w). The EP-vertices split the sum edges into subedges (b splits jk); likewise the PPP-vertices and the PP-edges (w splits ab, cd, and ef).

The sum polygons subdivide each other into polygonal regions, called facets, that are bounded by sub-edges. Some facets form closed surfaces, called shells, that divide space into open regions, called cells. The Minkowski sum consists of the cells with winding number one. Our example has two cells separated by one shell, and the bounded cell is the Minkowski sum (Fig. 1b). Each of the three sum polygons in Fig. 1c contributes one facet to the shell.

The Minkowski sum boundary is contained in the union of those sum polygons that pass a compatibility test, called the convolution. A convex vertex is compatible with a face if the exterior normal of the face is also an exterior normal for the vertex. Two convex edges are compatible if they have a common exterior normal. An exterior normal is a vector based at a boundary point of the polyhedron that defines a half-space whose intersection with the polyhedron is locally empty.

Fig. 1a shows a convex vertex v a convex edge e and a face f with exterior normals indicated by arrows.

Convolution-based Minkowski sum algorithms compute facets, shells, and cells from the convolution instead of from the full set of sum polygons. The convolution is a small subset of the sum polygons. Reducing the input size by a factor of d can reduce the computation time by a factor of d^3 because every triple of sum polygons can generate a PPP-vertex. In our example, the convolution comprises 4% of the 2672 sum polygons. Fig. 2 shows a larger example ($2 \oplus 5$ in Sec. 6) in which the convolution comprises 0.8% of 12,872,480 sum polygons. The convolution is rendered translucently to expose the interior sum polygons, which are drawn in blue. The outer shell is rendered translucently to expose the inner shells, which are drawn in green. It is sliced by a plane to show that there are no inner shells on the Minkowski sum boundary; the translucent part is on the far side of this plane.

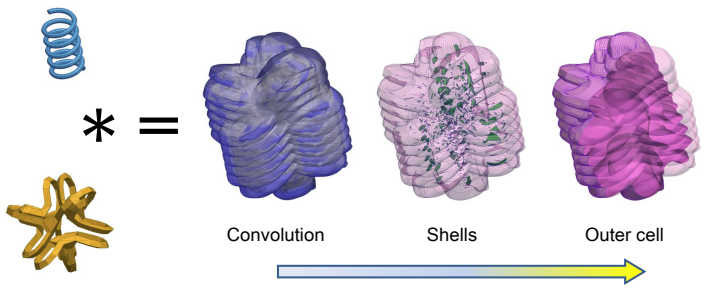


Figure 2: Minkowski sum of helix and knot.

Fig. 3 summarizes our Minkowski sum algorithm. Step 1 computes the convolution. Step 2 computes the PP-edges, step 3 computes the PPP-vertices, and step 4 computes the facets. Step 5 groups the facets into shells and cells. A shell bounds a Minkowski sum cell if $-A + v$ and B do not intersect with v an arbitrary vertex of the shell. The cells are obtained by computing the nesting order of these shells. The details appear in our prior paper [10].

Input: Polyhedra A and B.

1. Construct sum polygons.
2. Intersect sum polygons to obtain PP-edges.
3. Intersect PP-edges to obtain PPP-vertices.
4. Subdivide sum polygons into facets.
5. Form shells and cells.

Output: Cells of the Minkowski sum.

Figure 3: Minkowski sum algorithm.

4 Improved algorithm

We describe the parts of the improved algorithm that differ from the original [10, 2]. Some are algorithmic improvements and the rest are adaptions to GPU programming. The GPU implementation is described in Sec. 5.

4.1 Step 1: Sum polygon construction

Step 1 constructs sum polygons for the compatible vertex/face and edge/edge pairs. The CPU implementation accelerates the computation with a binary space partition of the Gaussian sphere. This data structure is ill-suited to the GPU because it requires global memory. The computing speed of the GPU allows us to enumerate all the candidate pairs. For each convex vertex of one polyhedron, a GPU thread tests compatibility with each triangle of the other polyhedron. For each convex edge of the first polyhedron, a thread tests compatibility with each convex edge of the second polyhedron.

4.2 Step 2: Sum polygon intersection

Step 2 finds the intersecting sum polygons by constructing a kd-tree and testing every pair of sum polygons in every leaf. A pair can appear in multiple leafs if some splitting planes intersect both sum polygons. Our prior algorithm uses a hash table to detect duplicate pairs. Hash tables require global memory and globally synchronized memory access among many threads, hence are ill-suited to the GPU.

We have developed a novel alternative that takes constant time per pair and that uses constant space per polygon. Each sum polygon p is labeled with a bit vector $l(p) = 0$. We construct a kd-tree for the labeled polygons. If the splitting plane at depth d intersects p, p is assigned to the left subtree and the right subtree is assigned a copy of p the dth bit of whose label is set to one. If not, p is assigned to one subtree. Sum polygons p and q in a leaf are tested for intersection if $l(p) \wedge l(q) = 0$ (the bitwise *and*).

Fig. 4 shows an example in which sum polygons p and q are inserted into a kd-tree with root n_0, depth-1 nodes n_1 and n_2, and leafs $n_3, \ldots, n_6$. Although the pair appears in n_3, n_5, and n_6, it is only tested in n_3 because $l(p) \wedge l(q)$ is 10 in n_5 and is 11 in n_6.

The proof that a pair of intersecting sum polygons is tested is by induction on the kd-tree depth d. For $d = 0$, both labels are zero. For $d > 0$, we show that instances of both polygons are assigned to the same subtree of the root and that the first bit is zero for one of their labels. The result follows by the inductive hypothesis. If both polygons are to the right of the cutting plane, both are assigned to the right subtree. If neither is to the right, both are assigned to the left subtree (and perhaps copies are assigned to the right subtree too). The first bit of both labels is zero in both these cases. Otherwise, one polygon is to the right of the cutting plane and so is assigned to the right subtree with first bit zero. The other cannot be to the left because the polygons intersect. Hence, it intersects the cutting plane and a copy is assigned to the right subtree.

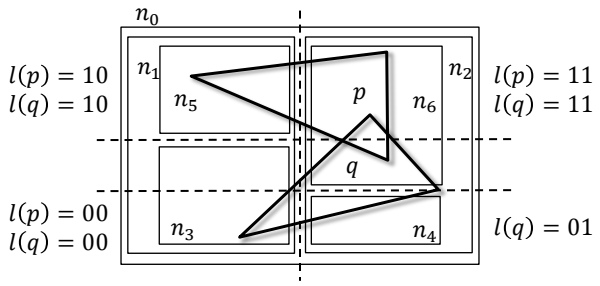


Figure 4: Labeled sum polygons p and q in depth-2 kd-tree.

The GPU implementation of our kd-tree construction algorithm builds upon a prior GPU kd-tree algorithm [11] that does not address duplicate pairs and that splits the polygons that intersect the cutting planes.

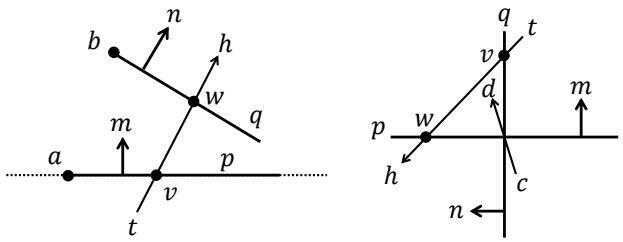
4.3 Step 4: Vertex sorting

Step 4 sorts the EP-vertices along their sum edges and the PPP-vertices along their PP-edges. Our prior algorithm orders vertices v and w along an edge $e = th$ with the predicate $(w-v)\cdot(h-t) > 0$. Evaluating this predicate dominates the running time of Minkowski sum computation. The cost is highest on nearly identical vertices because extended precision evaluation is required (Sec. 4.6). We have developed alternate predicates that express the order of v and w along e in terms of the relation between e and the vertices of the sum polygons p and q that it intersects at v and w. These vertices are well separated unless p and q are nearly coplanar. Thus, the alternate predicates resolve 99% of the nearly identical pairs using floating point arithmetic.

Suppose p and q are disjoint and one polygon, say q, lies on one side of the plane of the other polygon, p (Fig. 5a). Define $s_1 = m \cdot (b - a)$ and $s_2 = m \cdot u$ with $u = h - t$, a and b vertices of p and q, and m and n their normals. If $s_1 s_2 > 0$, e intersects p before q, so v precedes w. If p and q define a PP-edge $f = cd$, their planes define four quadrants that meet at f (Fig. 5b). The order of v and w along e is determined by the order in which e traverses the quadrants and by the orientation of e around f. Define $s_1 = ((d - c) \times u) \cdot (t - c)$, $s_2 = m \cdot u$, and $s_3 = n \cdot u$. If c is the intersection of q with a p edge with tangent x, define $s_4 = n \cdot x$; if the roles of p and q are reversed, $s_4 = -m \cdot x$. If $s_1 s_2 s_3 s_4 < 0$, v precedes w.

4.4 Step 4: Blocked subedges

The size of the Minkowski sum is proportional to the input size in practice and this is provable under mild input restrictions [12]. Yet the subdivision of the convolution can be far larger, as illustrated in Fig. 2. Our improved algorithm addresses this problem in step 4 during subedge construction. A vertex or a subedge is *blocked* if it is in the interior of the Minkowski sum. Blocked elements cannot contribute to the output. The improved algorithm removes blocked vertices and prevents the construction of blocked subedges, which also reduces the number of facets. We handle



(a) (b)

Figure 5: Disjoint (a) and intersecting (b) sum polygons.

two cases that account for almost all blocked elements and that are easy to compute.

Let v be an EP-vertex or a PPP-vertex that is the intersection of a sum polygon f with normal n and a sum edge or a PP-edge $e = th$ (Fig. 6). If $n \cdot (h - t) > 0$, the intersection of tv with a neighborhood of v is in the interior of the Minkowski sum by a standard result. Since the only way to leave the interior is by crossing the boundary, the subedge uv with $u \in [t, v)$ is blocked, so it is not constructed. Likewise for the subedge vw with $w \in (v, h]$ if $n \cdot (h - t) < 0$. In our example, only v_2v_3 is constructed. If both incident subedges of v are blocked, v is also blocked. If v is an EP-vertex that is an endpoint of a PP-edge f, the subedge, xv or vx, of f incident on v is not constructed. The subedges incident on x are also blocked, but are not worth computing.

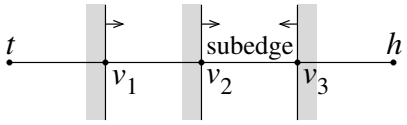


Figure 6: Blocked subedges.

The remaining memory bottleneck occurs in sorting the vertices along their edges prior to subedge construction. The GPU implementation eases this bottleneck by processing the edges in groups. As blocked vertices are removed in earlier groups, memory becomes available for processing the later groups.

4.5 Step 5: Shell formation

The CPU implementation of step 5 groups the facets into shells by breadth-first traversal. That algorithm is inefficient on the GPU because each cycle traverses one level of the tree, so the running time is proportional to the tree diameter independently of the number of processors. The GPU implementation uses a union-find algorithm to merge the facets in parallel. Each facet is initialized to a singleton tree. In each cycle, each facet is assigned a GPU thread that merges its tree with those of its neighboring facets. Since the number of merges is essentially constant (the inverse Ackermann function), the time complexity is proportional to n/p for n facets and p processors.

4.6 Robustness

The Minkowski sum algorithm is formulated in the real-RAM model where real arithmetic is exact and has unit cost. The control logic is expressed in terms of predicates: polynomials in the input parameters (the vertex coordinates of the polyhedra) whose signs are interpreted as truth values. The input is assumed to be non-degenerate, meaning that no predicate evaluates to zero. However, degeneracy is common in applications due to design constraints, symmetry, and repeated structures. The robustness problem is how to implement the algorithm accurately, efficiently, and for all inputs.

Robustness problems are typically handled heuristically in current practice, for example by treating small quantities as zero and by handling common degeneracies with custom logic. This approach becomes untenable as the input size grows and the algorithms become more complicated. The mainstream algorithmic robustness strategy is to evaluate predicates exactly using integer arithmetic [13]. The prior robust Minkowski sum implementations employ this strategy (Sec. 2). The disadvantages are high time and space complexity, and the need to handle degeneracy.

We prefer to evaluate predicates in floating point arithmetic because it is accurate, fast, and memory efficient. The robustness challenge is that even a tiny rounding error can cause a predicate to be assigned the wrong sign, which can create a large error in the algorithm output. Controlled perturbation [14] addresses this challenge by replacing the input with a perturbed input for which every predicate evaluation is correct and none is degenerate. The perturbation size δ bounds the backward error of the algorithm: the distance from the actual input to an alternate input for which the output is correct.

Our prior Minkowski sum algorithm uses a variant of controlled perturbation, called controlled linear perturbation [2]. It picks a random perturbation direction and a tiny initial δ. As it evaluates each predicate, it minimally increases δ to ensure that the predicate value is far enough from zero that its sign is correct despite rounding error. This robustness strategy has several drawbacks. It does not handle *singular* predicates whose value and gradient are zero, so we had to develop custom logic for them. The output error is tied to the number of degenerate predicates and to the magnitude of their first derivatives. We obtained unacceptable errors on some moderate-sized engineering inputs. Moreover, the correctness proof assumes that the linear term of each predicates dominates the higher order terms, which is impractical to verify. Finally, the algorithm employs global state variables, which reduces GPU efficiency.

The improved algorithm uses our adaptive-precision controlled perturbation (ACP) robustness strategy [3] to solve these problems. The user specifies the maximum output error and ACP provably enforces it. Singular predicates do not require custom logic, although they increase the running time. There are no global state variables.

The user specifies input parameters with values of type double float, which ACP perturbs by up to δ. The user defines other parameters as rational functions of prior parameters. ACP evaluates a predicate polynomial in interval arithmetic to obtain $[l, u]$, returns a sign of -1 if $u < 0$, and returns 1 if $l > 0$. If $l \leq 0 \leq u$, ACP reevaluates the predicate at a higher precision, using the MPFR multiple precision library [15]. Evaluation succeeds for sufficiently high precision if the predicate is non-degenerate, which is true with high probability because the input perturbation has

exponentially more possible values than the number of zeros of the predicate.

We adapt ACP to GPU programming as follows. The initial precision is float, rather than double float, because it is 3–4 times faster on the GPU, yet resolves 99% of the predicates. The next two precisions are double float and quad double float [16]. If a fourth level is required, the predicate is transferred to the CPU where it is evaluated with MPFR. Although we could use arbitrary precision on the GPU [16], the CPU calls are so rare that there would be no performance benefit.

5 GPU implementation

5.1 Data structure

We use a half-edge data structure to store input polyhedra and sum polygons in the GPU memory. A half-edge has attributes of head and tail vertices, an incident polygon, and a next half-edge. Since more than two sum polygons may share a half-edge, we add two more attributes, ccw-next and ccw-prev. The ccw-next is the next half-edge in counterclockwise order around the vector $(q - p)$, where p and q are the end vertices of the half-edge with the smaller and the larger indices, respectively. Each edge attribute is stored in a separate unsigned integer array. A vertex position is stored in a 4D vector of single precision numbers, *V*. The fourth component ($V[i].w$) stores a truncation error of coordinates to the single precision. For input vertices given in the single precision, it is set to 0. Due to the limit of GPU memory, vertex positions in the double precision and quad-double precision are not stored in memory, but computed on demand. For convolution data, the vertex position array stores sum vertices, EP-vertices and PPP-vertices in this order.

5.2 Sum polygon construction

Compatible feature pairs defining sum polygons are found among all candidate pairs from two input polyhedra. All convex vertex/triangle and convex edge/convex edge pairs are enumerated and sent to GPU threads to test the compatibility of the paired features. Compatible pairs are compactly stored into a global memory by using an atomic operation. An atomic operation allows GPU threads to write data one after the other, but is very slow. To reduce atomic operation calls slowing down the GPU, we divides the GPU threads into groups of 256 threads, and let each group temporarily store compatible pairs into its local shared memory. Each thread group calls an atomic operation once to get the next position of the global memory to flush out its compatible pairs.

Sum vertices are created from the compatible feature pairs. For each compatible feature pair, a GPU thread generates the pairs of its vertices making sum vertices of the corresponding sum polygon. We then sort the array of input vertex pairs, and eliminate redundant pairs among them. Each entry of the array represents a sum vertex ID. Two input vertex positions of each pair i is added in single precision with rounding-up mode and rounding-down mode, respectively. Let v_{ru} and v_{rd} be the results of them. The sum vertex position is stored as $V[i].xyz = 0.5(v_{ru} + v_{rd})$ and $V[i].w = 0.5||v_{ru} - v_{rd}||_\infty$.

Sum polygons are now constructed from the compatible feature pairs. A GPU thread stores sum polygon information of each feature pair, including the sum vertex IDs, the sum half-edge

IDs, and the polygon normal, into separate integer and single-float vector arrays. Identical sum half-edges from the sum polygons are linked together by setting the $ccw - next$ and $ccw - prev$ attributes.

5.3 Sum polygon intersection

5.3.1 Two polygon intersection

The kd-tree construction step yields an array of sum polygons contained in leaf nodes. The first polygon of each leaf node is indexed by another array. For efficient generation of polygon pairs, a group of m GPU threads cooperate for each leaf node. A thread group has its own temporary shared buffer to store intermediate polygon pairs before flushing out them to the global memory. A thread t enumerates index pairs $(a, b) = (i/n, i \mathrm{mod} n)$ where $i = t, \cdots, (n^2 - 1) + t$ with stepsize m and n is the number of polygons of the leaf node. For each pair with $a < b$, we test the bit-vectors of polygon a and polygon b and store the pair in the temporary buffer. If the pairs in the temporary buffer reach a limit, the thread group flushes out the pairs to the global memory. After all the thread groups have finished, the polygon pairs stored in the global memory are filtered with the oriented-bounding box test.

Exact polygon intersection is computed for the remaining polygon pairs. We construct an array of edge/polygon pairs by pairing an edge of one polygon with the other for each polygon pair. The array is sorted to remove redundant pairs, and then fed to GPU threads computing intersection. An intersecting edge/polygon pair (e, p) makes an EP-vertex whose position is stored in V after the sum vertices. (e, p) is also stored in a 2D integer array P_{EP} for higher-precision evaluation demanded later in the following steps. For each polygon pair p_a and p_b $(p_a \leq p_b)$ with two EP-vertices or one shared- and one EP-vertex, we create a PP-edge and also add (p_a, p_b) to an array, E_{PP}, which is sorted in the lexicographic order.

5.3.2 Three polygon intersection

To find three-polygon intersections, we sort E_{PP} and then segment it with respect to the first polygon ID. For each array segment s with the first polygon p_s, we generate polygon triples by combining the polygons intersecting p_s in the segment. A group of m GPU threads cooperates to generate polygon triples in a similar manner to the polygon pair generation step. A thread t enumerates index pairs $(a, b) = (i/n, i \bmod n)$ where $i = t + km$ for $k = 0, \cdots, (n^2 - 1)/m - 1$ and n is the number of PP-edges of the segment. For index pairs with $a < b$, a polygon triple (p_s, p_j, p_k) is formed from the a-th pair (p_s, p_j) and the b-th pair (p_s, p_k) in the array segment.

The polygon triples are fed to GPU threads, each of which computes intersection of three sum polygons. We choose two best PP-edges out of three PP-edges from the three sum polygons, and compute their intersection point. The two best PP-edges are those with the largest determinant of edge directions. For intersecting polygon triples, we create a PPP-vertex at the end of EP-vertices in V, and store the polygon triple and the three PP-edges in two 4D integer arrays, P_{3P} and E_{3P}, for higher-precision evaluation and polygon subdivision. $P_{3P}[i].w$ records the two PP-edges used for intersection computation.

5.4 Sum polygon subdivision

5.4.1 Vertex sorting

The EP-vertices and the PPP-vertices are sorted along their edges for edge subdivision. An EP-vertex i is paired with its sum edge $P_{EP}[i].x$, and a PPP-vertex j is paired with its three PP-edges, $E_{3P}[j].x$, y, and z, respectively. Let L be the edge/vertex pair array. So, there are $n_{ep} + 3 \times n_{ppp}$ vertex/edge pairs in total where n_{ep} is the number of EP-vertices and n_{ppp} is the number of PPP-vertices. The line parameter of each edge/vertex pair (e, v) is computed by a GPU thread as $d = (v - h) \cdot (h - t)$ for edge $e = (t, h)$. The edge/vertex array L is sorted in the increasing order of the lower bound of d, to find adjacent vertices with overlapping d.

A GPU thread attempts to sort a group of transitively overlapping vertices, using a selection sort with the alternate ordering predicates. Since we use a different predicate for each case in Fig. 5, we need to determine the case quickly. We construct an index table H whose entry i has the starting index of polygon pairs with p_i as the first one in E_{PP}. For two sum polygons p_i and p_j given ($i < j$), we look up (p_i, q_j) in $E_{PP}[H[i]]$ to $E_{PP}[h[i + 1] - 1]$ by binary search.

For the groups whose vertex order is not resolved by these predicates, we determine their order with quad-double arithmetic.

5.4.2 Blocked subedges

For each pair $(e, v) \in L$, the GPU computes the sign of $N \cdot (h - t)$ where N is the normal of the intersecting polygon at v and $e = th$. Let S be the array of the signs. Let B be a boolean flag array indicating blocked-ness of vertices and initially set to false.

We determine blocked vertices along each edge with two rounds of GPU execution. In the first round, a GPU thread determines $B[L[i].v] =$ true, if $L[i].e = L[i + 1].e$ and $S[i] > 0$ and $S[i + 1] > 0$, or if $L[i].e = L[i - 1].e$ and $S[i - 1] < 0$ and $S[i] < 0$. In the second round, a GPU thread sets $B[L[i].v]$ to true, if $L[i].e = L[i + 1].e$ and $S[i] > 0$ and $B[L[i + 1].v] =$ true or if $L[i].e = L[i - 1].e$ and $S[i] < 0$ and $B[L[i - 1].v] =$ true. A GPU thread i outputs a subedge $L[i].v$ $L[i + 1].v$ if $S[i] > 0$, $S[i + 1] < 0$, and $B[L[i].v] = B[L[i + 1].v] =$ false when $L[i].e = L[i + 1].e$. If $L[i].e \neq L[i + 1].e$, the GPU thread outputs two subedges made with $L[i].v$ and the head point of $L[i].e$, and $L[i + 1].v$ and the tail point of $L[i + 1].e$.

On large inputs, we divide a set of edges into several groups, and perform sorting and blocking on each group one by one. To reduce occupied GPU memory, we remove blocked PPP-vertices out of the vertex set as soon as they are found in each group. Blocked EP-vertices are not explicitly removed but just excluded from the edge/vertex array of subsequent groups, because they may be required to compute the sign of a PPP-vertex.

For further data reduction, a PP-edge with no PPP-vertices on it is deleted if one of its end points is a blocked vertex.

5.4.3 Subedge connection

Since we use a half-edge structure, each subedge e has to be stored as several half-edges with connection information. During the subedge connection step, a half-edge from e, incident to a

polygon f, is temporarily represented as a pair (f, e), and then inserted into the half-edge structure at the end.

In order to connect half-edges efficiently, we create an array T of triples (p, f, e), from half-edges (f, e) such that p is one end point of e. Half-edges incident to the same p on the same f are arranged together by sorting T with key (p, f). Since sum vertices, EP-vertices, and PPP-vertices have IDs increasing in this order, the half-edges with sum vertices come before those with EP-vertices after sorting, followed by those with PPP-vertices.

For triples with the same p and f in T, a GPU thread connects the half-edges related with them. The first case is that p is an EP-vertex and two subedges in the triples, e_1 and e_2, are from a sum edge and a PP-edge, respectively. Then, two half-edges (f, e_1) and (f, e_2) are connected appropriately. The second case is that p is a PPP-vertex, at which at most two subedges, e_1 and e_2, are non-blocked among the subedges in the triples. To find them, the GPU thread tests if the other end point of each subedge is blocked by an intersecting sum polygon. Two halfedges (f, e_1) and (f, e_2) are connected such that they make a left-turn chain. For other cases, the GPU thread rearranges its given subedges in counterclockwise order around p on f, and connects two adjacent half-edges if the intersecting sum polygon of one does not block the other and vice versa.

To maximize the GPU performance, triples of identical cases should be adjacent in T, because otherwise execution flow would be seriously divergent among adjacent GPU threads. Fortunately, the sorting above put the triples with identical cases together, except the EP-vertex cases. To separate triples of different EP-vertex cases during the sorting step, we negate f if the intersecting sum edge of EP-vertex p is incident to f. Then, these triples come before the triples of the other EP-vertex case after sorting. The GPU thread connecting subedges recognizes the EP-vertex case according to the sign of f.

5.4.4 Polygon subdivision

Subpolygons are formed by traversing half-edges originated from subedges. A GPU thread starts traversing from one half-edge every five in the half-edge array. The GPU thread follows connected half-edges along the next pointer. If the next pointer is NULL, the thread fails to find a loop bounding a polygon and stops. If the next pointer is the starting half-edge, a loop bounding a sub polygon is found. The half-edges on the loop are made incident to a new polygon, whose ID is set to the lowest ID among these half-edges. The reason to choose the lowest half-edge ID is that two different GPU threads may happen to trace the same polygon boundary asynchronously. Since the lowest half-edge ID on a polygon boundary is unique, the two threads consistently set the polygon ID to the same number. After all GPU threads have been done, we repeat tracing with untraced half-edges if there remains.

5.5 Shell formation

For shell formation, we use two arrays, *parent* and *manifold*. *parent*$[f]$ has the parent node of f in the tree representing a set containing f. If f is a root, *parent*$[f] = f$. The root of the tree is found by tracing back through the *parent*, and all the *parent* entries are reset to the root after

tracing to lower the tree height. *manifold* $[f] = true$ indicates that the neighborhood of f is a locally manifold surface.

Each polygon f initially forms a singleton set by setting *parent*$[f] = f$ and *manifold* $[f]$ =*true*. A GPU thread takes each polgyon f, and merges the set containing f with the sets of its adjacent polygons. For a half-edge $e = th$ incident to f, let e' be *ccw-next*$[e]$ if $t < h$, or *ccw-prev*$[e]$ otherwise. If e' does not exist for an edge of f, the thread sets *maniford*$(f_0) = false$ where f_0 is the root of the set containing f. Otherwise, we merge the two sets containing f and f' by setting *parent*$[f'_0] = f_0$ if $f_0 < f'_0$, or *parent*$[f_0] = f'_0$ otherwise.

After all GPU threads have finished, we scan the polgyon list to see if there exist any adjacent polygon pairs not in the same sets. If there are, we repeat the merge process again for them. In practice, such adjacent polygon pairs are rarely found after the first merging.

6 Results

We tested our Minkowski sum algorithm on nine polyhedra (Fig. 7): 1. frame (30 triangles), 2. knot (992), 3. torus (2,068), 4. dragon (2,328), 5. helix (4,012), 6. bull (12,396), 7. inner ear (32,236), 8. horse (36,964), and 9. sphere (760). We tested all 45 pairs of polyhedra. The 36 pairs of distinct polyhedra are generic because the parts are dissimilar; the 9 identical pairs are highly degenerate because every sum polygon is duplicated. The ACP δ is 10^{-8}. The CPU tests are on one core of an Intel Core 2 Duo. The GPU tests are on an Nvidia GTX580 with 3GB of memory, and use CUDA.

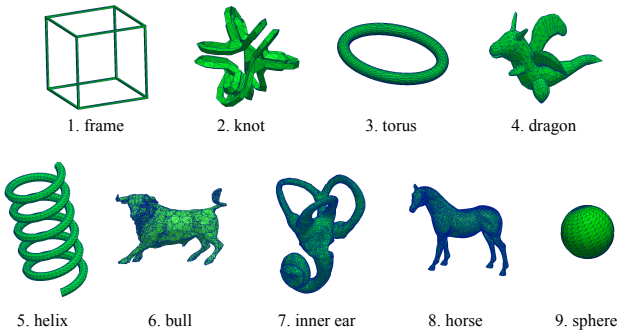


Figure 7: Test polyhedra.

Fig. 8 shows ten representative pairs and Table 1 shows their results. The speedup factor f excludes the running time for testing if $-A + v$ and B intersect in step 5 of Fig. 3. Including this time would unfairly increase f because the CPU implementation uses a naive intersection algorithm whereas the GPU implementation uses a sophisticated algorithm [17]. However, the intersection time is included in the running time t for completeness. We obtain speedups of 20–40 on the first nine tests. The speedup increases to 62 on the tenth test, which is far larger than the others. We chose the $1 \oplus 6$ and $1 \oplus 7$ tests for comparison with the volumetric approach [9]. Our program is 25 faster on these examples; we estimate that it would be 10 times faster on their GPU.

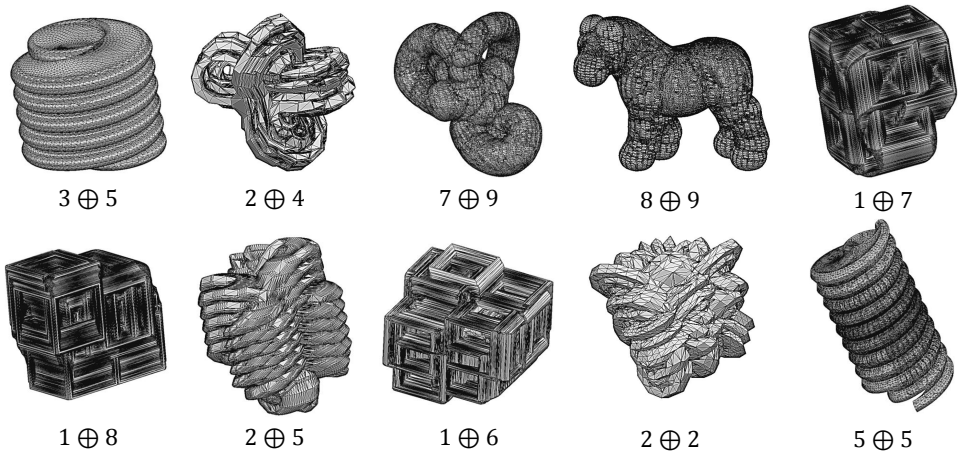


Figure 8: Minkowski sums.

Table 1: Results: i input, p sum polygons in convolution, e PP-edges, v EP- and PPP-vertices, r percentage non-blocked subedges, s and m subdivision and Minkowski sum complexity (vertices plus edges plus faces), c polyhedron intersection tests, t running time on GPU in seconds, and f speedup factor.

i	p	e	v	r	s	m	c	t	f
$3 \oplus 5$	$47,294$	$55,886$	$74,898$	22	$173,537$	$93,598$	0	0.2	23
$2 \oplus 4$	$56,561$	$161,509$	$217,452$	24	$168,891$	$62,600$	47	0.3	27
$7 \oplus 9$	$77,474$	$81,293$	$103,218$	30	$293,391$	$190,872$	0	0.3	33
$8 \oplus 9$	$87,603$	$101,195$	$127,792$	30	$321,555$	$207,788$	1	0.4	29
$1 \oplus 7$	$51,519$	$291,466$	$466,408$	11	$194,986$	$87,246$	92	0.3	28
$1 \oplus 8$	$66,640$	$414,331$	$646,368$	10	$247,081$	$117,978$	30	0.4	29
$2 \oplus 5$	$108,230$	$669,797$	$988,761$	21	$332,966$	$49,012$	2196	0.7	38
$1 \oplus 6$	$37,520$	$501,574$	$862,895$	8	$128,410$	$66,640$	113	0.5	21
$2 \oplus 2$	$48,341$	$715,740$	1.8e6	5	$110,950$	$30,854$	864	0.9	32
$5 \oplus 5$	$228,001$	5.2e6	2.6e7	22	1.3e6	1.1e6	22	5.2	62

More importantly, our error is 10^{-8} and is under user control, while theirs is 10^{-3} and is determined by the GPU memory capacity.

The CPU program generates accurate outputs for all 45 pairs. The full results appear in Table 2. The six pairs $6 \oplus 6$, $6 \oplus 7$, $6 \oplus 8$, $7 \oplus 7$, $7 \oplus 8$, and $8 \oplus 8$ are the hardest tests by far. They have 2.4–3.9 million sum polygons, 10–24 million PP-edges, and running times of 500–1200 seconds. The GPU program achieves an average speedup factor of 30 on the other 39 pairs. The speedup increases with input size; for example, the five pairs with CPU running times greater than 100 seconds have an average speedup of 61. The GPU program exceeds its memory capacity on the six hardest pairs. We estimate that 15GB of memory are required for these pairs based on the data from the CPU solutions.

7 Discussion

We have presented a robust algorithm for Minkowski sums of polyhedra that uses the adaptive-precision controlled perturbation (ACP) robustness library. We have evaluated it on 45 tests of various complexities and have showed that it outperforms all prior work. We conclude with plans for future work.

The evaluation shows that the limiting factor in GPU computation is memory capacity. We reduce memory use by removing blocked vertices and sub-edges. We employ data partitioning in this computation because it is the memory bottleneck. To handle larger inputs, partitioning must start earlier, ideally when the convolution is constructed. The challenge is to determine balanced partition blocks based on a predicted data distribution. Recent work [18] provides a faster GPU program for testing if two polyhedra intersect, but the cost of this step is already small.

Another approach to large inputs is to distribute the algorithm over multiple GPU's. We can already distribute sorting of vertices along edges, which is the computational bottleneck in the current implementation. Advances in partitioning will allow us to distribute PP-edge and PPP-vertex construction.

Another research direction is to compute swept volumes using an extension of the convolution algorithm. We aim to adapt our CPU implementation [2] to the GPU.

Acknowledgments

Kyung supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology (2012-0001704). Sacks supported by NSF grant CCF-0904832. Milenkovic supported by NSF grant CCF-0904707.

Table 2: Results: i input, p sum polygons, e PP-edges, v EP- and PPP-vertices, r percentage non-blocked subedges, s and m subdivision and Minkowski sum complexity, t running time in seconds, f speedup factor.

$1 \oplus 1$	821	8,730	25,698	7.2	1,870	1,562	0.2	2
$1 \oplus 2$	5,060	78,168	155,805	22.0	49,241	37,106	0.2	6
$1 \oplus 3$	4,146	15,958	24,645	23.0	26,865	19,918	0.2	3
$1 \oplus 4$	5,796	23,601	29,871	27.1	26,451	18,510	0.1	6
$1 \oplus 5$	10,983	103,923	222,490	26.1	147,620	92,970	0.2	13
$1 \oplus 6$	37,520	501,574	862,895	7.7	128,410	66,640	0.5	21
$1 \oplus 7$	51,519	291,466	466,408	11.5	194,986	87,246	0.3	40
$1 \oplus 8$	66,640	414,331	646,368	9.9	247,081	117,978	0.4	27
$1 \oplus 9$	1,256	872	856	50.8	5,253	4,904	0.1	2
$2 \oplus 2$	48,341	715,740	1.8e6	5.1	110,950	30,854	0.9	31
$2 \oplus 3$	32,478	51,649	69,713	33.1	106,890	41,902	0.2	14
$2 \oplus 4$	56,561	161,509	217,451	23.9	168,891	62,600	0.3	25
$2 \oplus 5$	108,230	669,797	988,761	20.6	332,966	49,012	0.7	39
$2 \oplus 6$	353,858	3.6e6	7.0e6	13.9	728,552	295,704	2.6	66
$2 \oplus 7$	502,576	2.3e6	3.6e6	18.7	1.2e6	265,320	1.9	68
$2 \oplus 8$	579,745	2.0e6	3.0e6	19.9	1.6e6	390,574	2.0	69
$2 \oplus 9$	13,163	9,345	10,404	45.9	53,023	32,116	0.1	9
$3 \oplus 3$	16,366	108,731	189,818	11.3	58,490	47,056	0.4	9
$3 \oplus 4$	31,150	27,372	31,968	41.6	110,125	34,954	0.2	12
$3 \oplus 5$	47,294	55,886	74,898	22.1	173,537	93,598	0.2	23
$3 \oplus 6$	168,190	305,165	419,108	27.7	509,507	184,324	0.6	32
$3 \oplus 7$	144,660	234,618	328,757	28.9	513,688	204,878	0.6	36
$3 \oplus 8$	167,010	257,592	349,451	30.5	604,132	286,672	0.8	39
$3 \oplus 9$	4,908	1,580	1,560	66.8	19,526	18,764	0.1	9
$4 \oplus 4$	68,405	191,361	278,402	26.1	235,257	106,200	0.5	23
$4 \oplus 5$	108,747	144,777	175,211	38.8	378,700	89,114	0.4	22
$4 \oplus 6$	371,722	723,996	987,427	27.9	1.0e6	282,494	0.9	38
$4 \oplus 7$	467,350	681,339	894,569	31.7	1.5e6	315,968	1.2	37
$4 \oplus 8$	550,794	678,350	859,852	33.7	1.8e6	493,322	1.3	39
$4 \oplus 9$	13,598	9,476	10,758	43.0	49,164	17,422	0.1	9
$5 \oplus 5$	228,001	5.2e6	2.6e7	1.3	1.3e6	1.1e6	5.2	62
$5 \oplus 6$	609,721	1.9e6	3.0e6	23.6	1.4e6	310,600	1.8	41
$5 \oplus 7$	681,322	1.8e6	2.7e6	25.1	1.9e6	333,210	2.0	52
$5 \oplus 8$	763,875	1.5e6	2.1e6	27.4	2.4e6	449,598	2.0	50
$5 \oplus 9$	24,392	14,883	15,668	47.4	103,073	56,134	0.1	16
$6 \oplus 6$	2.7e6						failed	
$6 \oplus 7$	2.9e6						failed	

i	p	e	v	r	s	m	t	f
$6 \oplus 8$	3.5e6						failed	
$6 \oplus 9$	74,013	106,642	137,002	30.5	252,196	124,890	0.3	23
$7 \oplus 7$	2.3e6						failed	
$7 \oplus 8$	2.6e6						failed	
$7 \oplus 9$	77,474	81,293	103,218	30.4	293,391	190,872	0.3	30
$8 \oplus 8$	3.9e6						failed	
$8 \oplus 9$	87,603	101,195	127,792	29.7	321,555	207,788	0.4	31
$9 \oplus 9$	2,983	1,003	1,034	46.0	11,510	9,796	0.1	4

References

[1] A. Kaul, M. A. O'Connor, Computing minkowski sums of regular polyhedra, Tech. Rep. RC 18891, IBM Research Division, Yorktown Heights (1993).

[2] E. Sacks, V. Milenkovic, M.-H. Kyung, Controlled linear perturbation, Computer-Aided Design 43 (10) (2011) 1250–1257.

[3] V. Milenkovic, E. Sacks, S. Trac, Robust complete path planning in the plane, in: Proceedings of the Workshop on the Algorithmic Foundations of Robotics, WAFR, 2012, pp. 37–52.

[4] P. Hachenberger, Exact minkowski sums of polyhedra and exact and efficient decomposition of polyhedra into convex pieces, Algorithmica 55 (2009) 329–345.

[5] M. Campen, L. Kobbelt, Polygonal boundary evaluation of Minkowski sums and swept volumes, Eurographics Symposium on Geometry Processing 29 (5) (2010) 1613–1622.

[6] J. Lien, A simple method for computing minkowski sum boundary in 3d using collision detection, Algorithmic Foundation of Robotics VIII (2009) 401–415.

[7] J. R. Shewchuk, Adaptive precision floating-point arithmetic and fast robust geometric predicates, Discrete and Computational Geometry 18 (1997) 305–363.

[8] G. Varadhan, D. Manocha, Accurate minkowski sum approximation of polyhedral models, Graphical Models 68 (4) (2006) 343–355.

[9] W. Li, S. McMains, a GPU-based voxelization approach to 3D minkowski sum computation, in: J. Keyser, M.-S. Kim (Eds.), Solid Modeling, ACM, 2010, pp. 31–40.

[10] V. Milenkovic, E. Sacks, M.-H. Kyung, Robust Minkowski sums of polyhedra via controlled linear perturbation, in: Proceedings of the 14th ACM Symposium on Solid and Physical Modeling, ACM, 2010, pp. 23–30.

[11] K. Zhou, Q. Hou, R. Wang, B. Guo, Real-time kd-tree construction on graphics hardware, ACM Transactions on Graphics 27 (5) (2008) 126:1–126:11.

[12] A. F. van der Stappen, D. Halperin, M. H. Overmars, The complexity of the free space for a robot moving amidst fat obstacles, Computational Geometry Theory and Applications 3 (1993) 353–373.

[13] C. Yap, Robust geometric computation, in: J. E. Goodman, J. O'Rourke (Eds.), Handbook of discrete and computational geometry, 2nd Edition, CRC Press, Boca Raton, FL, 2004, Ch. 41, pp. 927–952.

[14] D. Halperin, Controlled perturbation for certified geometric computing with fixed-precision arithmetic, in: ICMS, 2010, pp. 92–95.

[15] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, P. Zimmermann, MPFR: A multiple precision binary floating point library with correct rounding, ACM Transactions on Mathematical Software 33 (2007) 13:1–13:15.

[16] M. Lu, B. He, Q. Luo, Supporting extended precision on graphics processors, in: Proceedings of the Sixth International Workshop on Data Management on New Hardware, DaMoN '10, ACM, New York, NY, USA, 2010, pp. 19–26.

[17] S. Gottschalk, M. C. Lin, D. Manocha, Obbtree: A hierarchical structure for rapid interference detection, in: Proceedings of SIGGRAPH '96, ACM, 1996, pp. 171–180.

[18] J. Pan, D. Manocha, Gpu-based parallel collision detection for fast motion planning, The International Journal of Robotics Research 31 (2) (2012) 187–200.